Verification of Device Drivers and Intelligent Controllers: A Case Study

David Monniaux
CNRS
Laboratoire d'informatique de l'École normale supérieure
45, rue d'Ulm
75230 Paris cedex 5, France
David.Monniaux@ens.fr

ABSTRACT

The soundness of device drivers generally cannot be verified in isolation, but has to take into account the reactions of the hardware devices. In critical embedded systems, interfaces often were simple "volatile" variables, and the interface specification typically a list of bounds on these variables. Some newer systems use "intelligent" controllers that handle dynamic worklists in shared memory and perform direct memory accesses, all asynchronously from the main processor. Thus, it is impossible to truly verify the device driver without taking the intelligent device into account, because incorrect programming of the device can lead to dire consequences, such as memory zones being erased.

We have successfully verified a device driver extracted from a critical industrial system, asynchronously combined with a model for a USB OHCI controller. This paper studies this case, as well as introduces a model and analysis techniques for this asynchronous composition.

Categories and Subject Descriptors

D.2.4 [**Software engineering**]: Software/Program Verification—*Correctness proofs, Formal methods, Validation*; D.4.5 [**Operating systems**]: Reliability—*Verification*; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs—*Invariants, Mechanical verification, Specification techniques*

General Terms

Verification, Reliability

Keywords

Device driver, verification, USB, OHCI, asynchronous, direct memory access, linked lists, parallelism

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

Safety-critical systems consist of some application program (control/command, surveillance...) and some system code. Norms (such as DO-178B [12] for avionics) establish a hierarchy of levels of safety requirements, depending on the severity of the consequences of a failure of the system. Systems at the highest levels typically do not include a full operating system, but rather a limited basic input/output subsystem.

In the past, we have successfully verified the absence of runtime errors in several classes of safety-critical programs [7, 3, 2]. These programs interfaced to the outside world through memory-mapped input/output registers. Even if external co-processors were used, such as for efficient digital signal processing (DSP), they sent and received data through a shared memory bank. No synchronization was needed; the only hypothesis was that reads and writes of the data words passed through such interfaces were atomic.[1]

Such systems could be verified with as simple an input/output specification as: reads from integer registers can yield any value in the corresponding integer type; reads from floating-point DSP fields can yield any value in a certain user-specified interval (because the DSP cannot output values outside this interval). It is also possible to ask the analyzer to verify that outputs are within certain bounds.

There is now growing pressure, even in safety-critical industries, to move from proprietary or specialized busses to off-the-shelf components used in personal computers (such as the *Universal Serial Bus*, USB) or derived from them (such as the AFDX bus, derived from Ethernet). These busses were originally not designed for safety-critical systems. Both the protocols and the host controllers (interface circuits) used are orders of magnitude more complex than the simple ones previously used:

- They implement features such as hot-plugging, plug-and-play, dynamic reconfiguration etc. that are unneeded in most safety-critical systems, which have a static design. Even if those features are unneeded and unused, they result in complex initialization and pro-

[1] Atomicity is essential even for such a simple setup. If a 32-bit word is written to shared memory in such a way that the write is split into two 16-bit writes, there is a chance that the reader will see inconsistent data. For instance, if a value goes from 0 to -1 (0xFFFFFFFF in hexadecimal, 2-complement), there may be a brief instant during which the value may be 0xFFFF0000 or 0x0000FFFF. The same applies if the read is non-atomic.

gramming techniques.[2] It is in general impossible to alter the controller design to get rid of such undesirable functionality.

- They have higher bandwidth and thus implement efficient transfer techniques: direct memory access (DMA) and worklists. Thus, each host controller essentially behaves like an asynchronous process modifying the memory of the main system.

In order to ensure that a safety-critical system does not encounter runtime errors ("crashes", invalid operations, arithmetic overflows, invalid array or pointer accesses, data corruption caused by the intelligent controllers...), it is thus necessary to verify not only the application code, but also the device drivers, and even the device drivers composed asynchronously with the "intelligent" host controllers.

We thus have verified the driver code interfacing a critical system with a USB OpenHCI [5] host controller. OpenHCI, or OHCI, is a specification for how USB 1.1 host controllers should be programmed[3] There exist a variety of chips, or IP blocks,[4] implementing that specification. The specification leaves significant leeway as to possible behaviors of the implementation.

The driver was imposed to us as a C program. It is not an academic example, and uses many features that often cause problems for software analysis, such as pointer address computations and deeply linked data structures. However, this driver is simpler than drivers found in general purpose operating systems, since these have to handle dynamic reconfigurations (e.g. plugging in a mouse in the middle of operations). Since the embedded system targeted has a static layout (peripherals are plugged into known positions), data structures are simply and statically allocated in arrays. Still, since the USB controller uses linked lists for its worklists, the analysis had to cope with dynamic data structures (that is, data structures were pointers are dynamically updated).

The OHCI specification is informal, written in English, and we thus had to developed a formal model for it. Because we intended the industrial end-users to be able to understand and modify the specification, we also wrote this model in the C language as an asynchronous process. This program, however, is not a software implementation of OHCI: since what matters for verification is not to model precisely the behaviors of the system, but only a super-set of these behaviors, we left out many aspects such as timing delays and bandwidth constraints. Our host controller simulator often uses non-deterministic choices (for instance, at any moment, the current transmission may be aborted and an error reported), which would rule it out as a concrete simulator.

The interface between the controller and the driver does not operate according to the principles of shared memory programming that are usually advocated in the software engineering literature. There are no mutual exclusion primitives (e.g. semaphores, mutexes); there only exist coarse-grained flags for blocking certain operations of the controller, but the interface is designed so that these should be used minimally. The correctness of the system relies on the very weak hypothesis that 32-bit aligned read and writes to non-cacheable shared memory are atomic and not reordered.

[2]The one controller feature that caused us significant problem for verifying the driver we considered, the *done queue*, was actually unused by the driver, but there was no way to turn it off in the controller. Thus, needless, complexity in the controller can result in much harder analysis.

[3]There exists another specification for USB 1.1 controllers, known as UHCI, and a specification called EHCI for USB 2.0 controllers. Linux users may check for the `{e,o,u}hci_hcd` kernel modules, where `hcd` stands for *host controller driver*.

[4]IP blocks are off-the-shelf designs for parts of a custom microchip.

2. SINGLE-THREAD ANALYSIS

We have extended the ASTRÉE static analyzer in order to perform that analysis. ASTRÉE was initially designed solely for single-threaded code; however, in order to adapt it for the form of parallelism exhibited by the systems considered in this paper, we leveraged several of the techniques already used for single-threaded code. We shall thus describe briefly the single-threaded analysis, and refer the reader to the bibliography for details.

ASTRÉE [7, 3, 2] is a static analyzer originally designed for verifying safety properties of large-scale critical control/command programs as found in e.g. avionics. Such software contains many floating-point computations and digital filters, but use of pointers was limited to passing by reference and other similar simple uses. Pointer computations (that is, computing on addresses as arithmetic objects) were initially prohibited.

ASTRÉE works as follows. It represents sets of memory states using symbolic techniques (for instance, bounds on numeric variables). A set of memory states (or, more generally, execution traces) S is *abstracted* by a symbolic value $S^\sharp$, as long as $S \subseteq \gamma(S^\sharp)$. The collecting semantics of a program fragment P maps a set of memory states S to another $[\![S]\!](S)$. The analyzer will not attempt to compute this semantics exactly (because of undecidability or very large state space issues) but rather to map an abstraction $S^\sharp$ of the input environment to an abstraction $[\![S]\!]^\sharp(S^\sharp)$ of the output environment.

The analysis must not "forget" some reachable states, thus the *soundness condition*

$$[\![P]\!] \circ \gamma(S^\sharp) \subseteq \gamma \circ [\![P]\!]^\sharp(S^\sharp). \quad (1)$$

This inclusion relation reads as follows: if one starts in a state abstracted by $S^\sharp$ (a symbolic representation of a set of initial states) and executes the program P, then, for all possible states s' reachable at the end of the execution of P, s' should be abstracted by $[\![P]\!]^\sharp(S^\sharp)$, the result of the symbolic (or "abstract") execution of P over $S^\sharp$. We say that a state s is abstracted by $S^\sharp$, noted $s \in \gamma(S^\sharp)$, if the symbolic set of states denoted by $S^\sharp$ contains s. For instance, if $S^\sharp$ includes an interval constraint $\mathtt{x} \in [a, b]$, then any s abstracted $S^\sharp$ must verify $s(\mathtt{x}) \in [a, b]$, where $s(\mathtt{x})$ is the value of variable `x` in state s.

The analyzer also flags for any "abstractly reachable" state that results in a runtime error or other violation of user-specified assertions; warnings are then issued. Since the analysis is not exact, *false alarms* — warnings about problems that cannot happen in reality — may be issued. These are inevitable, since it is impossible for any analysis method to be both *sound* (warn about all possible problems) and *complete* (no false alarms), barring hypothesis such as a finite state space (and not a large one, because of complexity reasons).

Most of the complexity inside the ASTRÉE analyzer concerns the analysis of floating-point computations, which is outside the scope of this paper The tool was later extended to be able to analyze programs doing pointer arithmetic. Pointers are modeled as *base*+ *offset*, where the *base* is a tag for a variable in the source code, and the *offset* is counted from the start of the memory block containing the variable. This is important, since many of the structures used by the driver and controller are in a single very large compound variable mapped in a non-cacheable memory space.

Offsets are analyzed using non relational abstract domains: intervals $[m, M]$ and congruences $a + b\mathbb{Z}$; the former model the *bounds* of the pointer and the second the *stride* (if a pointer addresses 16-byte structures, the stride will be 16). A special domain representing finite sets of integers is used to abstract irregular access patterns (accesses that do not have a well-defined stride). Offsets may also be constrained using relational domains, e.g. octagons [9].

For better analysis of controllers, we added a domain that keeps track, for each integer variable, which bits of this variable can be 0 or 1; this is because hardware features are often turned on and off by single bits inside command or status words and the analysis has to take these into account.

The analyzer implements dynamic trace partitioning [8]: instead of analyzing a code fragment for all possible input states, it splits the input states according to some predicates (the number of iterations in a loop, the value of a variable, or an arbitrary test) and analyzes the code fragment separately in each context. Consider the following program:

```
if (condition) {
  a;
} else {
  b;
}
c;
```

Partitioning traces on the first step means that the continuation `c` of the test will be analyzed separately in each context, as though the program were:

```
if (condition) {
  a;
  c;
} else {
  b;
  c;
}
```

Partitioning is useful in many circumstances where a complex, often specialized, relational domain would be useful. Earlier work on ASTRÉE described examples where analysis of floating-point computations is much simpler if one e.g. partitions one test, such as the sign of a variable. [8][sec. 1] A common occurrence is when one has a pointer, or an array index, that may point to a variety of locations that each verify a local invariant: that is, there exists for all values of `i` a certain relationship between, say, `x[i]` and `y[i]`, but this relationship is not true if one takes `x[i]` and `y[j]` for $\mathtt{i} \neq \mathtt{j}$. If we do not partition, then our numeric abstract domain must reflect this relationship. By partitioning according to the value of `i`, we can use a much simpler numeric domain. [8, sec. 4.2] This partitioning feature was important for the analysis of parts of the driver that perform updates to parts of data structures that have strong relationships, e.g. pointers defining the beginning and the end of the same data block: if `p` points to a structure with two fields `begin` and `end`, then it is better to perform the analysis for all possible points-to values of `p` and keep relationships between `p->x` and `p->y`.

One of the most delicate issues in program analysis, and in ASTRÉE, is automatically inferring loop invariants. Consider the following loop:

```
while (condition) {
  code;
}
```

Leaving aside techniques such as loop unrolling and partitioning, this loop is analyzed by computing an approximation of the set of reachable states at the beginning of the loop, at point `A`:

```
A:
  if (! condition) goto B:
  code;
  goto A;
B:
```

The exact set is the least fixpoint of $X \mapsto X_0 \cup [\![code]\!](X \setminus [\![condition]\!])$. In order to compute an over-approximation of this set, ASTRÉE uses widenings. [6, sec. 4.3]. It chooses an ascending sequence of abstract sets and test if these are invariants, using the analysis as feedback (and stops once an invariant is obtained).

The output of the analyzer is a list of program locations and possible runtime errors. Errors that are detected are:

- out-of-bounds array access;
- null pointer and incorrect pointer dereferencing;
- invalid pointer arithmetic;
- arithmetic overflow, division by zero, etc.

3. MODELING AND ANALYSIS OF PARALLELISM

We describe the behavior of the controller as the disjunction of many atomic actions, such as: writing a 32-bit word to shared memory, loading a new pointer from shared memory, etc. Each of these actions is active only if certain guard conditions are met, such as: a bit in a control register is set, or two pointers are different.

These actions represent a coarse over-approximation of what the controller can actually do. In an actual controller, there exist a variety of sequentiality constraints: some action is possible only after another has taken place, as well as timing constraints: because of bandwidth limitations, transfers cannot happen instantly. However, a fine-grained model of the controller would be difficult to write, difficult to decompose into atomic actions, and would unduly complicate the analysis. Furthermore, it seems very reasonable that the correctness of a driver does not depend on fine timing constraints on an external device.

We group all atomic actions of the controller in a single atomic action a, which is essentially the nondeterministic choice between all possible actions constrained by the guards.

Let p_i be the atomic steps of execution of the main program . A trace of execution of the program composed with

the controller is a sequence of steps $p_1a^*p_2a^*p_3a^*$; any number of a steps can be performed between each atomic step of the main program. We could analyze the program in this model, however this would be very expensive.

As with many other parallel program analysis techniques, we implement a form of *partial order reduction*. Program steps and controller steps do not interfere unless the program touches the shared memory. As a consequence, we only consider traces of the form $p_1p_2a^*p_3a^*$ where each a^* precedes a step p_i such that p_i reads or writes shared memory. The shared memory zones are known from the documentation of the driver and can be specified in the configuration file of the analyzer. It is possible to check the correctness of these indications by checking that the controller never reads or writes outside of its own private variables and the indicated zones; however, this feature is not implemented in our prototype.

Astrée analyzes the main program (application code and USB driver) as though it were single-threaded, thus considering "p" steps. However, before any memory access, it checks whether it concerns shared memory. If so, the analyzer does the equivalent of analyzing the following loop:
`while(random_choice) { a }`
through an approximate fixpoint computation, before analyzing p.

4. THE USB CONTROLLER AND DRIVER

Now that we have described the analysis, we shall now describe the system to be analyzed, consisting of a controller and a driver.

4.1 OHCI Controller

We shall here summarize the most salient points of the OHCI host controller specification. Each host controller communicates with the main processor through several channels [5, §3.3.2]:

- a memory-mapped register bank
- a shared-memory area known as HCCA
- linked lists of *endpoint descriptors* (ED) and *transfer descriptors* (TD), occasionally building trees (in some sense),
- various zones for direct-to-memory (DMA) transfers, containing the data read or written to USB devices.

The main difficulty, with respect to program analysis, are the singly-linked lists of ED and TD. These lists are concurrently updated by the driver and the host controller. An *endpoint* is some kind of functions attached to devices (for instance, a USB sound card will have an endpoint for transmitting sound samples, another for mixer data, etc.). Endpoints can be of four types (control, bulk, interrupt, isochronous). Each endpoint is described by an ED, and to each ED is attached a TD list as in Fig. 1.

In general, there is no need to alter the endpoint lists unless a device is added or removed. In embedded systems of the kind considered here, there is no "hotplugging" and in fact the position on the busses of all devices is known at the design stage, thus the endpoint list can safely be initialized once and for all.

Each TD describes a chunk of data yet to be sent or received by the controller. The TD lists are are modified whenever the driver wishes to send or receive USB data. Because

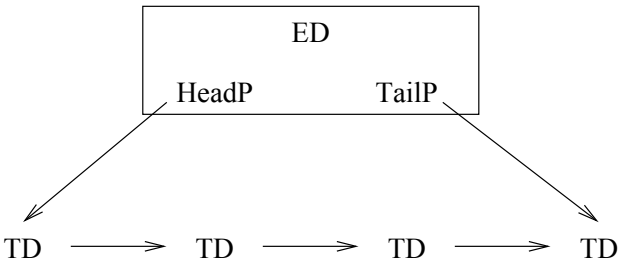

Figure 1: Each TD points to the head and tail elements of its transmission list. The last element is a "dummy".

transmission requests are a frequent event, the OHCI interface allows adding new requests to the end of the TD lists with no mutual exclusion mechanism, except the assumption that a 32-bit write is atomic. Each ED has a pointer to the head and tail of the associated TD list (Fig. 1). The last element of the list is a "dummy" element, the controller never attempts transferring it. In order to add a new request to the transmission list, one has to fill the required information in the "dummy" tail TD, allocate a new "dummy" TD, make the tail TD point to it, then advance the tail pointer to the new "dummy" element through an atomic write.

The controller processes TDs in sequence. For each TD, it transfers the corresponding data from or to main memory using direct memory access (DMA), asynchronously from the main processor. Each TD contains start and end address for a DMA data buffer. The controller signals successful or unsuccessful completion by positioning appropriate flags inside the TD. The controller transfers completed TDs to a "done queue", another singly-linked list: the "next" field of the newly completed TD will point to the last element moved to the done queue (obtained through the done queue pointer), and the done queue pointer will then point to the newly completed TD.

There exists two types of TD: generic TDs (16-byte long) and isochronous TDs (32-byte long). Isochronous TDs are used for isochronous endpoints, while generic TDs are used for the four other types.

4.2 Main Program

The main program is of the following form:

```
driver_initialization();
device_initializations();
application_initialization();
while (TRUE) {
  wait_for_clock();
  application();
  driver(); /* processing of TD lists */
}
```

After the driver and the USB controller are initialized, the driver initializes the various devices through various types of control messages, and checks whether they correctly respond. Then, the system enters a synchronous loop.

At each clock tick, the driver monitors its various TD lists, possibly detects errors, and initializes new TDs. Because this is a sensitive embedded system, no dynamic memory allocation is used; the various lists are allocated in arrays, and pointer arithmetic is used to address the elements.

The only interaction of the application with USB is to read and write elements in the data buffers. Since the layout of the system is fixed, it knows exactly which endpoint sends which data and can directly find the right buffer. These buffers are sent to the devices or received from them using DMA. There are no function calls from the application to the driver (and neither in the reverse direction); we thus have been able to ignore the application when analyzing the driver.

5. SPECIFICATION AND ANALYSIS

We wrote a 400-line simulation of a OHCI controller. This simulation is essentially a series of non-deterministic choices between the various atomic actions that can be performed. For the sake of simplicity of design, and also of simplicity of communication with industrial partners, this simulation is also written in C, but makes liberal use of nondeterministic choice primitives. It would thus not be usable as a concrete simulator of a controller, but is suitable for analysis. The whole of the controller actions is summarized in a single procedure. Because there are three controllers in the system, there are three calls to that procedure, passing different register areas as parameter.

The simulation works as follows: if the controller is busy transferring a data block, then it can choose between transferring the next 32-bit word in the data block, signaling an error, or moving to the next data block (and thus the next TD). If the TD is finished, then it can move to the next ED. Note that error signaling is possible at any moment (it is always possible that the controller signals a hardware malfunction, for instance some wire could have been cut or unplugged), thus it would be impossible to prove the correct transmission of data using this simulation.

The main program is given as C source code, approximately 3000 line long. The relationship between the driver and the controller is specified in a configuration file: shared memory zones are listed, with the name of the simulation procedure. The analysis concerns both the initialization phases and the regular runtime.

The main goal of the analysis is to prove that neither the driver nor the controller will transfer data incorrectly (pointers outside allowed memory ranges etc.). Such incorrect transfers could lead to the driver and application crashing. The analysis, however, does not verify that transfers are properly programmed: this would involve modeling the devices, the USB protocol and issues such as bandwidth, which is beyond the capabilities of our analysis tool.

The main limitation to that approach is the difficulty of specifying the controller as a set of atomic actions. The specification of the controller given to us is an English-language technical manual. As with other informal specifications, there are delicate issues, such as the instant at which certain memory accesses take place, that can be misunderstood. Thus, it is not obvious that the formal specification (as a nondeterministic program) encompasses all the behaviors permitted by the English specification (not counting whether or not the actual hardware implementation of the controller fits the English specification!).

We had to slightly alter the driver and host controller specification so as to adapt them to what Astrée can handle. The driver occasionally used `goto`'s in order to signal errors, in a way that Astrée does not analyze precisely due to limitations in the partitioning heuristics; we slightly rewrote the code (better partitioning could achieve the same results). In addition, Astrée, though it can handle some untyped manipulations in programs (such as: taking a pointer, storing it into an integer, then taking the integer as a pointer), could not cope with the practice of the OHCI specification of putting information is the two low-order bits of pointers supposed to be aligned on 4-byte boundaries. We removed that information and the associated bit-masking operations. Finally, we removed some "padding" in some data structures, which caused Astrée to introduce a great number of useless variables. We feel that neither of these changes substantially altered the substance of the verification performed, and are just the results of minor limitations of Astrée.

The main difficulty in the analysis of the system is the "done queue". In simple embedded contexts, such as the one we studied, there will be typically one static pool of TD elements per ED, and thus, a priori, all TD lists should be separate, leading to a simple, non-relational, separation invariant: each ED points into the corresponding TD pool, and each element in a TD pool may point only into the same TD pool. However, the OHCI controller, after completing the transmission of a TD, moves the TD to a global "done queue", breaking this separation: the "next" pointer of the TD no longer points to the next TD in the processing list, but to the last element inserted into the "done queue", which may be from a different ED. There still is a separation property, but it is a complex, dynamic, one: at any moment, the set of elements reachable from the list head associated with an ED is in a single TD pool, but elements from that TD pool may also be in the done queue. An element at the same memory location will move from one queue to another. Ironically, the "done queue" is never used by the driver that we were shown.

The driver submitted to us could however be proved correct (that is, proved that it cannot exhibit runtime errors) even in the presence of the done queue. The analysis was incapable of inferring complex dynamic separation properties, thus, due to the building of the "done queue", it confused 16-bit generic TDs and 32-bits isochronous TDs: after a while, all TD pointers used by the driver were deemed capable of pointing to most other used TD pointers. It did not matter because all TD data structures had been padded to the same alignment by the driver's designers; thus, even if the analysis took into account some spurious behaviors (processing of 16-bit TDs as though they were 32-bit), these did not introduce false alarms.

Analysis is completed in 10 hours on a 2 GHz PC, using a few hundred megabytes of memory (in comparison, disabling the "done queue" results in an analysis time of 6 minutes). In either case, there are no alarms, meaning *no possible runtime errors*, and no false alarms. The absence of true alarms is not surprising: even though the controller implements some sophisticated data structures (dynamically rearranged linked lists), the driver has a conservative, simple design, and has been well tested before. We expect that more complex developments, with more transfer modes, will need more advanced pointer analyses.

6. RELATED WORKS AND CONCLUSION

We have shown that it is possible to prove the absence of runtime errors in simple (yet real) drivers that interact with active, "intelligent" devices, even with shared pointer variables. To our knowledge, all earlier works had focused

either on toy systems, either ignored the controller. As a consequence, some bugs due to active controllers may have stayed unnoticed.

The importance of checking device drivers in addition to higher level code has been recognized by practitioners from industry. According to Microsoft, in 2003, 85% of recently failures in Windows XP came from the drivers [13]; it is also reported that for certain kinds of errors (lock/unlock), Linux device drivers contain as much as seven times the rate of errors as the main kernel code [4].

There exist two different approaches towards the issue of bugs in device drivers. The one that we considered is *program verification*: the analysis tool outputs sound results; that is, assuming sound modeling of the hardware, if the analysis tool lists no possible runtime errors, none can happen at runtime. However, soundness often comes at a steep price: the analysis may be too costly, or it may refuse to run on some programs. As a consequence, *bug-finding* methods have been proposed: the goal is not to prove the absence of errors in the system, but to efficiently find some of the existing bugs.

Bug-finding methods typically exclude parts of the system (other threads, intelligent devices...) that are difficult to analyze; even with single-threaded code, these techniques might be unsound. The point is to obtain, in practice, lists of warnings that, for the most part, are "true" warnings, corresponding to problems that can really happen, as opposed to problems that cannot happen but are warned against as the result of the imprecision of the analysis. This list, however, needs not be exhaustive; that is, it is acceptable that some true errors may be ignored, if trying to find them all would come at the expense of listing many "false" warnings. Such techniques target non-critical code such as the operating system and the applications typically used on personal computers, where some infrequent crashes and other dysfunctions are accepted. Bug-finding tools should thus concentrate on the "low-hanging fruit" first. Bug-finding methods have been proposed for device drivers:

- The SLAM group at Microsoft develops tools based on model-checking of boolean abstractions. [1] Initially, the boolean abstraction contains only the control-flow of the driver and boolean variables, and then it can be refined using additional predicates. The tool verifies that the device drivers interact correctly with the application programming interface (API) of the operating system; that is, issues like never freeing the same buffer twice, or locking and unlocking the same lock in alternation (but not locking it twice in a row). It does not model "intelligent" external devices — it does not model concurrency with shared memory at all —, it models integers as $\mathbb{Z}$ "ideal integers" and not a fixed-width bit vectors and can miss errors due to overflow, and does not check memory safety at all — in fact it assumes that the code contains no "wild pointers". Thus, not only is their tool unsound, but also it ignores the issue that we model in this paper, that is, the interaction between a device driver and an asynchronous device. This tool is, however, capable of scaling up to real-life system, and is now used commercially as the *Static Driver Verifier*.

- Some less formalized, more ad-hoc techniques have been applied with great success for finding bugs in Linux and OpenBSD kernel code [4]. Techniques used may be statistical, may attempt detecting "inconsistent" usage, etc. The goal of such heuristics is not to provide programmers with some insurance that the program functions well, but to quickly and automatically point to probable bugs in the code.

Because the verification of C programs, with all the semantic oddities and pointer arithmetic, is difficult, some have advocated replacing C with a "safe subset" (e.g. MISRA C [10]) or some type-safe language related to it (e.g. CCured [11]). Both of these approaches are likely to be defeated by the way that hardware controllers are designed. These approaches essentially advocate getting rid of "unsafe" constructs, either in a heuristic way (industrial "safe subset" guidelines) or in a more formal way, founded in program semantics. The foremost unsafe constructs are the juggling of pointers, with unsafe conversions to and from arithmetic types, pointer arithmetic, etc.; however, these are largely imposed by the hardware design.

The object of this paper is a *verification* technique. The goal is to prove the absence of errors in critical code, and finding bugs is only a secondary objective. Critical code tends to be simpler than desktop or server operating systems, and to be operated in a more constrained environment. For instance, the layout of the system is generally fully known at the design stage and thus no dynamic "hotplug" facility is needed. One of the bugs listed in a Windows driver [1, §6] occurs if a connector is unplugged at the same time that the operating system performs a *close* request; this example is emblematic of race conditions that may be difficult to reproduce and occur in rarely exercised code paths. Such complexities are avoided if there is simply no code for dynamic reconfiguration, no dynamic data structures, etc. Critical code is written according to stringent guidelines [12]; also, because there are fewer cases to test, the code will probably have been thoroughly tested; therefore there is little hope of finding many bugs if the analysis tool is applied at the end of the development process (but a bug-finding tool may come handy to speed up testing and development). Thus, the goal of the verification tool is to find the very last bugs, and to prove the absence of bugs.

Future work in that area should include:

- Better efficiency. The use of partitioning, which is very costly, should be limited. Even though some partial order reduction is already used, we think the number of interleavings considered is still too large.

- Efficient relational pointer abstract domains (e.g. for analyzing dynamic linked lists) should also be developed. These domains should be compatible with real-life low-level programming language, e.g. allow pointer arithmetic. We expect that future work for analyzing such kind of mixed systems (driver and intelligent controller) should include some better pointer shape analysis techniques than the ones that we used.

From a software engineering point of view, we propose methodological changes in the development of checking of system software used in safety-critical system. In our experience, it is never the case, except for very simple programs, that analysis succeeds the first time. The specification may be incorrect, the analyzer may be imprecise; there may be

abstract domains to add, or simply some abstract transfer functions that can be made more precise. When the analyzer issues warnings, one has to find their origin. Of course, it is better in that circumstance if the analyzer supplies backward analysis results (ASTRÉE does not yet do so). However, in practice, one also has to have an intuition of what the program is doing, and the kind of invariants that it relies on, so as to be able to check whether the analysis results fit expected behaviors and to refine the analysis if necessary. This suggests that analysis should be performed in collaboration with the designers of the embedded system, who presumably know well what should happen, at the time when the system is designed. This contrasts with the usual testing and bug-finding approaches, where generally the testers are kept separate from the developers so that they can have a fresh mind and even attempt things that "should be working". However, in the case of program analysis, this separation is counter-productive; the verification team will essentially have to reverse-engineer the program before attempting to refine analysis.

7. REFERENCES

[1] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys*. ACM, 2006.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer-Verlag, 2002.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[4] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88. ACM, 2001.

[5] Compaq, Microsoft, National Semiconductor. *OpenHCI Interface Specification for USB*, 1.0a edition, 2006.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13):103–179, 1992.

[7] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, number 3444 in Lecture Notes in Computer Science, pages 21–30, 2005.

[8] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, number 3444 in LNCS. Springer-Verlag, 2005.

[9] A. Miné. A few graph-based relational numerical abstract domains. In M.V. Hermenegildo and G. Puebla, editors, *SAS*, Lecture Notes in Computer Science 2477, pages 117–132. Springer-Verlag, September 2002.

[10] MISRA: The Motor Industry Software Reliability Association. *MISRA-C:2004 Guidelines for the use of the C language in critical systems*, October 2004.

[11] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139. ACM Press, 2002.

[12] RTCA / EUROCAE. *DO-178B / ED-12B: Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[13] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, pages 207–222. ACM, 2003.